OBJECT-ORIENTED COMPUTER PROGRAMMING

Adapted from a forth coming manuscript by:

Casey G. Cegielski, PhD
Assistant Professor of MIS
College of Business
Auburn University
Auburn University, AL 36849

Introduction

Over the last 10 years, object-oriented programming languages (OOPL), primarily C++ and Java, gained significant acceptance among and widespread utilization by computer programmers (Kolling, 1999). Today, these languages, principally because of their unique properties, are popular choices as development tools for computer business applications (Blaschek G., Pomberger, G., and Tritzinger, A., 1989). Generally, object-oriented programming languages are different from familiar procedural programming languages (PPL), such as COBOL and BASIC for example. The following sections of this paper define and describe the unique aspects of object-oriented programming languages as the languages conceptually differ from traditional procedural programming languages.

An Overview of Object-Oriented Programming Languages

In a traditional procedural programming language, a computer program is a set of interdependent procedures operating on data in the services of a particular goal whereas in object-oriented programming languages, a program is a set of autonomous objects that exchange data to fulfill a unified purpose (Pennington, Lee, Rehder, 1995; Ferrett and Offutt, 2002). Functionally, OOPL evolved from the established logic used in procedural languages – top-down modularity (Henderson, R. and Zorn, B, 1993). However, the collective purpose of those individuals developing early OOPL was to create computer programming languages that exhibited more modularity through less *coupling*, interdependency between program routines, and greater *cohesion*, the extent to which program routines are linked in purpose rather than function, than traditional procedural languages (Henderson, R. and Zorn, B, 1993). Highly coupled program routines are indivisible because each routine is dependent upon the function of another routine

(Kolling, 1999). A computer program that exhibits a high degree of coupling is more difficult to maintain, to extend, and to reuse due to extensive internal interdependences among routines (Kolling, 1999). Thus, the developers of OOPL sought to develop a mechanism through which singular procedures (programming routines) are separable and thus independent in scope, from an aggregate program of which they were only a component. The functional solution develop to minimize coupling in OOPL is *encapsulation*.

Encapsulation

Encapsulation, the packaging of programming code into wholly independent, self-contained units, is the basis of all OOPL (Kolling, 1999). Object oriented programmers accomplish the task of encapsulating code through the placement of programming routines in *classes*. This task, while simply stated conceptually, is much more difficult to apply practically. The difficultly many OO programmers experience when attempting to encapsulate program code is deciding which routines specifically require individual encapsulation (Kolling, 1999). Obviously, the extreme application of the theory of encapsulation suggests that a programmer encapsulate every routine in an individual class. This is neither always necessary nor always desirable. Thus, a programmer coding in an OOPL must distill, via cognitive abstraction from the general program purpose, a specific functional class schema that best supports a loosely coupled and highly cohesive program. The abstraction process, or object-oriented modeling, is both a skill and an art that requires an individual to posses a holistic view of the general use of the program in a context (i.e. the business function the program will support within a firm) and a specific understanding of the concepts of OOPL.

Inheritance

Minimization of coupling in OO programs is not the only benefit OO programmers derive through encapsulation of computer program code. Additionally, encapsulation provides OO programmers with the capacity to reuse, or inherit, previously developed encapsulated code. In OOPL, *inheritance* provides a mechanism for organizing and structuring software programs based on the functionality of existing

classes (Cho and Kim, 2002). To illustrate the concept, assume that an OO programmer intended to create a new end-user interface for a real-time inventory system. Many of the components of the new interface, such as buttons, text fields, and labels, already exist in the OO language library. Thus, the OO programmer would simply extend the previous classes from the library that contain the required objects (button et al.) rather than re-code new buttons, text fields, and labels for the interface. In OOPL, inheritance is a simple yet powerful application of encapsulation.

Polymorphism

Polymorphic expression, when combined with inheritance, is a particularly powerful aspect of OOPL that creates flexibility when reusing previously encapsulated code. In OOPL, polymorphism is the capability of an object to retain a generalized purpose while assuming different forms of application (Cho and Kim, 2002). For example, an "employee" class contains a method called "earnings." Assume the computation of employee wages is the general purpose of the earnings method. While each employee may earn wages differently (i.e. hourly pay rate or salary), the general earnings method in the employee class defines what is earnings. Any class that inherits the earnings method from the employee class could specifically define the actual computation of earnings for an individual object.

The fundamental concepts of all OOPL are the aforementioned characteristics – encapsulation, inheritance, and polymorphism. Although these ideas, in theory and practice, are interdependent, their application allows OO programmers the capacity to reduce program coupling and create autonomous, reusable, and flexible program code.